AFRL-RI-RS-TR-2016-028

SPECIFICATION IMPROVEMENT THROUGH ANALYSIS OF PROOF STRUCTURE (SITAPS): HIGH ASSURANCE SOFTWARE DEVELOPMENT

BAE SYSTEMS

FEBRUARY 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-028 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
MARK LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
FEBRUARY 2016	FINAL TECHNICAL REPORT	SEP 2013 – SEP 2015

4. TITLE AND SUBTITLE

SPECIFICATION IMPROVEMENT THROUGH ANALYSIS OF PROOF STRUCTURE (SITAPS): HIGH ASSURANCE SOFTWARE DEVELOPMENT

5a. CONTRACT NUMBER
FA8750-13-C-0240

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Howard Reubenstein, Greg Eakman, Tom Hawkins

5d. PROJECT NUMBER
HACM

5e. TASK NUMBER
SS

5f. WORK UNIT NUMBER
IA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

BAE Systems
6 New England Executive Park
Burlington, MA 01803

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-028

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2016-0232
Date Cleared: 22 JAN 2016

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Formal software verification methods and tools have made significant progress in their ability to model software designs and prove correctness theorems about the systems modeled. General adoption of these techniques has had limited penetration in the software development community. Two interrelated causes may account for barriers to adoption. First, many tools prove properties about models of the system as opposed to the actual implementation. Software engineers ultimately need to produce performant software implementations and therefore they are primarily concerned with properties of their implementations. Second, while it is crucial that formal derivation processes do not introduce deviations from the specification (or vulnerabilities) – a domain independent requirement – engineers also need to verify application and domain specific properties in building their implementations. The SITAPS (Specification Improvement Through Analysis of Proof Structure) project described in this report explores techniques that can be used to obtain greater domain and application specific assurances.

15. SUBJECT TERMS

High Assurance Software, Verification and Validation, Formal Methods, High Assurance Programming

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**STEVEN DRAGER**
U	U	U	UU	24	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

Table of Figures

Table of Tables

1 Summary

A formally verified specification (and correctly derived implementation) may produce undesirable behavior if the specification does not properly capture the required system behavior. Flaws, such as over-specification, under-specification, and incorrect assumptions about the environment lead to implementations that admit emergent system behavior. Ensuring that a specification properly captures desired behavior is achieved by validation. Validation addresses the improvement of a specification to reflect requirements and the system environment. The Specification Improvement through Analysis of Proof Structure (SITAPS) research effort developed proof-based validation techniques for improving the quality of formal specifications and for maximizing the effectiveness of formal verification efforts.

Use of individual formal reasoning tools is currently a complex undertaking. Few engineering practices or metrics exist around managing verification efforts, managing collaborative verification efforts (as the DARPA High-Assurance Cyber Military Systems (HACMS) program is exploring), or integrating verification with system construction practices. SITAPS strengthens specifications by providing tools to analyze a system's proofs and proof structures and to identify critical assumptions. These tools have been integrated with assurance case representations to extend their utility to semi-formal and information assurance arguments.

Cyber-attacks can invalidate reasonable assumptions leaving even a verified system to operate based on emergent behavior. Through derivation of proof metrics such as "assumption criticality" or "theorem root set size" SITAPS detects potentially brittle verification cases. SITAPS provides tools and techniques that can be used to obtain greater domain and application specific assurances which increases system resiliency and developer confidence in the appropriateness of the runtime behavior of a verified system.

2 Introduction

Programming involves building larger constructs out of smaller ones, using compositional design. How such pieces fit together is determined by a set of "rules": the syntax of the chosen programming language, the types of its constructs, what is allowed at runtime, and of course, the desired behavior. However, existing composition and checking mechanisms do not fully express the semantics of specific problem domains.

Formal methods offer stronger guarantees by proving theorems that state precisely how program constructs may be meaningfully combined—ruling out programs outside these patterns. However, the required level of expertise is significant and proving a significant set of properties of a program can increase development costs well over an order of magnitude.

The question is how to improve the understanding of program semantics in existing engineering shops, with an aim to bringing behavior more fully under the aegis of defined specifications. How do we make semantics a first class citizen? An additional challenge lies in making this a cost effective enterprise, since achieving complete assurance today requires unrealistic up-front

expenditures of expertise and time (despite data indicating that the cost of early error detection is often exceeded by required, late-in-cycle maintenance)[1].

Without addressing the semantics problem, software will continue to be written that turns on its creators, so to speak. It is not enough to know what it can do, as to know what it will do under every circumstance. Proof offers a solution, but only if we solve the problem of cost—and in a way that makes proof as obvious an ingredient of success as testing now is.

Formal software verification methods and tools have made significant progress in their ability to model software designs and prove correctness theorems about the systems modeled. General adoption of these techniques has had limited penetration in the software development community (though specific techniques have gained adherents, e.g., the use of model-checking to verify properties of finite-state implementations). Two interrelated causes may account for barriers to adoption.

First, many tools prove properties about models of the system as opposed to the actual implementation[2]. Software engineers ultimately need to produce performant software implementations and therefore they are primarily concerned with properties of their implementations. One approach taken by the formal methods community to assist with this need is to develop systems that use correctness preserving techniques to produce implementations that are provably equivalent to the specification they are derived from.

The second barrier is that generated code is rarely performant enough or interoperable with concrete software architectures. Software engineers need greater control over the code and need to author critical portions of the software themselves. Furthermore, while it is crucial that compilation-like processes do not introduce deviations from the specification (or vulnerabilities) – a domain independent requirement – engineers need to verify *application and domain specific* properties in building their implementations (e.g., the speed output control variable never requests a speed greater than 120).

Development of high assurance systems requires developing confidence in three distinct aspects of system development:

1. Correctness preserving implementation of the system specification (assurance in depth)
2. Development of a specification that meets (application specific) security policies (assurance in breadth)
3. Validation of the specification to domain and application requirements

The SITAPS effort is aimed at producing tools and techniques to improve the overall level of confidence in a system implementation as a whole by addressing items 2 and 3 in the above list

[1] In some cases the situation is pathological since management tools like earned-value are inconsistent with large upfront costs that do not result in a linear output of useful product (i.e., running – even if broken - code) - "Managerial Issues for Consideration and Use of Formal Methods," Studolph and Whitehead, FME 2003

[2] As reported again recently in: http://news.mit.edu/2015/crash-tolerant-data-storage-0824

(item 1 is well addressed by existing verification tools and correctness preserving compilers). Specification flaws can be introduced in a number of ways. SITAPS focuses on flaws introduced by the use of assumptions in the specification. Specification validation multiplies the effectiveness of the verification effort by ensuring that the specification captures appropriate behavioral definitions.

Assurance in depth is the focus of a significant body of work in both proof-based verification of correctness and computer security work including exploration into provably correct compilers like CompCert[3], secure host computers like SAFE[4], and host protection strategies like address space layout randomization. These sorts of technologies are application independent and focused on assuring that the compute host behaves according to its idealized specification and excludes aberrant behaviors associated with, e.g., buffer overflow or code injection attacks. These research areas provide the crucial foundation for building secure systems on secure platforms.

The software verification process (described above as "assurance in depth") has been a focus of the DARPA High-Assurance Cyber Military Systems program. When we began considering **assurance in breadth** we were lead towards consideration of less formal software development artifacts known as *assurance cases.* SITAPS includes assurance cases to help reason about the correctness of the specification itself and to provide a structured explanation that can be analyzed for complexity and dependencies.

A key motivation for the SITAPS effort was also to address the third aspect of high assurance system development, **validation**, by identifying and analyzing assumptions in a specification. For example, suppose an automobile safety system makes an application dependent assumption that no obstacle can approach at faster than 150 MPH relative velocity and further suppose this assumption is relied on by a number of system components. Identification and validation of this assumption is crucial to developing a high assurance system, particularly since the system will exhibit some sort of emergent behavior if put in a situation where the assumption is invalid. The system development question then becomes whether you want to rely on the unknown emergent behavior or whether the specification should be extended to handle this case specifically (e.g., at least in some fail-safe manner).

3 Methods, Assumptions, Procedures

SITAPS's approach is based on analysis of verification work products (produced as part of the DARPA High-Assurance Cyber Military Systems program) as a validation step to provide specification critique and improvement. While the effort required to formally verify complex component and system properties (as undertaken by HACMS performers) is considerable, our goal is to develop tools to extract, analyze and expose weakness in the developed proof artifacts. The insight we are applying is that checking a proposed solution to a problem is usually significantly easier than generating the solution. The goal of our feedback is to ensure that

[3] CompCert: http://compcert.inria.fr/
[4] SAFE: http://www.crash-safe.org/

verification efforts more accurately capture overall system requirements and that the system will work reliably under all conditions in the target environment.

During the initial stages of the project we reviewed proof activities with a number of the HACMS performers. While there were a diversity of artifacts and proof systems being used, except for the Rockwell-Collins AGREE system, there were not readily available extracts of the proof structure from the tools being used. For example, while Coq proves properties it does not dump an explanation of the proofs in any currently supported form.

The use of proof in HACMS components and systems was different than expected as compared, e.g., to the use of proof in mathematics. The proofs we discovered were typically provided by a successful application of the reasoning tool, often guided by a proof outline (tactics). The evidence of success is largely that the proof assistant terminates with a positive result.

In looking at extraction of proof structures we observed the following kinds of issues:

- A Computational Logic for Applicative Common Lisp (ACL2) does not store the steps of its proofs, only the results. ACL2 uses heuristics and a large rule base to attempt to prove theorems, and these heuristics make it difficult to instrument the capture of the proof structure.

- The XML export of Coq proofs was no longer supported (and the internals of ACL2, while open source, was just as difficult to extract, given the heuristics of the ACL2 proof assistant). It has not currently been possible to run examples of either of these proof structures through the SITAPS tools for analysis.

However, work done by Joosten[5] to extract and transform ACL2 libraries into Prolog provides a possible indirect way to extract information about the proof structure, although not the proof structure directly. ACL2 provides a partial list of theorems, definitions, and axioms used during a proof. Joosten was able to use raw Lisp mode to extract the rules supporting a proof in order to transfer the complete list to the Prolog environment. Of course, the individual steps were lost, but the new theorem proving environment would reproduce the steps, using the rule list as a guide.

Due to these sorts of impediments, we focused our validation efforts on two distinct experiments:

First, we created a pedagogical specification example based on the hotel room locking example from Daniel Jackson's Alloy book[6]. We developed a formalization of the example, stated properties of the design, proved theorems about the design, captured the verification in an assurance case, and ran the SITAPS metrics extraction tools on the resulting artifacts.

[5] Joosten, Sebastiaan, Kaliszyk, Cezary, and Urban, Josef, "Initial Experiments with TPTP-style Automated Theorem Provers on ACL2 Problems", F. Verbeek and J. Schmaltz (Editors), ACL2 Workshop 2014.
[6] Alloy: http://mitpress.mit.edu/books/software-abstractions

Hotel room locks and card keys use a simple protocol to manage the transition of rooms from one guest to the next. The lock maintains a code, remembering the last key to unlock the door. The key contains two codes, a previous code and a current code. The lock's code is initially the current code of the current guest, and the lock opens when its code matches a key's current code.

When a guest checks out, the lock still retains that guest key's code. A new guest checks in and gets a card with a new current code, and the previous code set to the previous guest's current code. The first time the new guest unlocks the door, the lock compares its code against the current code and fails, since it does not match. The lock then checks its code against the key's previous code. Since the key's previous code matches the locks code, the lock recognizes this key as the new guest, opens the door, and updates its code to match the current code of the new guest's key.

Second, we worked with one of the HACMS high-assurance Domain Specific Languages (DSLs), Ivory – developed by Galois, and created a verification capability for compiler assertions generated by the Ivory tools. We ran the verifier against test cases and small code samples provided by Galois. Verifying these assertions at design time increases the overall reliability of the system.

Ivory is a DSL in Haskell for embedded programming developed under the HACMS program. Ivory has semantics similar to C, but also provides memory safety, which is enforced by the Ivory type system. To capture design intent, Ivory has user specified assertions and procedure contracts. In addition, the Ivory compiler generates assertions to guard a program against a host of runtime violations including floating point exceptions, numerical overflows, index casting, and unbounded loops.

Verification of Ivory assertions is crucial to verifying correct behavior of Ivory-based components. Any assertion that is not verified becomes a run-time assertion that must be monitored during execution and handled if violated.

4 Results and Discussion

Validating specifications via extraction of assumptions from DARPA's HACMS component proof structures has proven significantly harder than we expected. In general, the reasoning tools that support high assurance *software* verification and development do not produce the kind of explanation traces (including assumptions) that can be captured and reasoned about. Tools like Coq and ACL2 implement a precise logic and can be guided by user input regarding reasoning strategies; however, they do not provide output of the types of explanations that are useful in tracing reasoning dependencies.

During development of the SITAPS extraction and analysis tools and incorporation of assurance cases[7] as part of the input argumentation structure (in addition to proof structure) we realized that assumptions govern not only the boundary conditions of system operation but they also govern expected **domain dependent** specifications (in the breadth) of normal operation and/or conditions in which making formal verification arguments may exceed state-of-the-art capabilities. We further realized that there appears to be a significant gap in the application of formal methods technology in that a (perhaps inordinate) focus of research activity is on domain independent correctness preserving techniques. This observation and our work with Ivory on proving code assertions (similar to assumptions) forms the basis of the broader recommendation coming out of SITAPS, i.e.:

> "The application of formal methods technologies to the verification of domain/application dependent code level properties is a high-value area for research advances and can advance the adoption of formal methods techniques by practicing software engineers. There is an opportunity to focus directly on the software engineer's fundamental problem of proving application specific properties of the code base they are developing. Adoption of formal methods techniques requires directly addressing the software engineer's task of producing high assurance executable systems with unique application requirements.[8]"

4.1 Assumption-Based Analysis[9]

SITAPS aims to strengthen the claims about a system's security, reliability, correctness, or other critical properties. Claims are based on proofs using formal models of the system, or less formal arguments called assurance cases, which structure arguments similar to the way a lawyer would argue a case (or a risk analyst would perform a hazard analysis). For cyber-physical systems, both assurance cases and formal proofs rely on assumptions about the environment in which these systems operate. The software environment includes the target platform, network (including cyber threats), sensors, and actuators. The systems engineering view of the environment would also include all things with which the sensors and actuators interact.

One premise of SITAPS is that the verification effort of a complex high-assurance system is itself an effort on par with the underlying software development effort and that it merits (requires) tools to manage the proof process. SITAPS tools focus on the capture of the proof or assurance case structure into a graphical database that encodes a general Goal Structuring Notation (GSN)[10] representation and then supports analysis of dependencies in the graph (independent of the logic used) focusing on management of assumptions.

---

[7] Integrating with the Certware assurance case tools: http://nasa.github.io/CertWare/
[8] Or more simply: provide more support for assurance in the breadth in a context useful to software engineers producing code.
[9] Reported in more detail in CDRL A010: Specification Analysis and Metrics Report
[10] GSN: http://www.goalstructuringnotation.info/

Assurance cases can be used to integrate verification arguments for multiple system components and subsystems. Each component may have its own assumptions and its own verified claims. The assurance case ties together individual arguments into an aggregate argument and may bridge reasoning gaps that while compelling are not amenable to formal theorem proving.

During creation of the hotel example an interesting case of argument decomposition occurred. The assumptions required in reasoning about one component (the lock/key combination) were discharged by proofs developed for another component (the hotel desk / key generator). One specific example is the assumption that "current and previous key codes are not duplicates" being discharged by a proof that the "key code generator does not produce successive duplicate key codes."

The informal argument that ties the two together is the assertion that the only source of room keys are those generated by the hotel front desk (the explicit modeling of which would likely lead to modeling of a more realistic source of hotel keys, e.g., production of counterfeit keys by unapproved duplication devices).

This observation leads to a proposed augmentation of Goal Structuring Notation to include explicit argumentation assumption discharge links.

ACL2 Proofs

We developed many types of proofs on the hotel example. Low level proofs addressed infrastructure semantics such as how the heap program state model works. Mid-level lemmas covered design and implementation properties, such as facade functions that do not affect the state of the system and read-only functions such as looking up the room number of a guest. Application level proofs of functional properties, covering the rules of keys opening locks and how new keys are generated depend heavily on the low level semantic proofs, and on the mid-level lemmas to ease the proof effort.

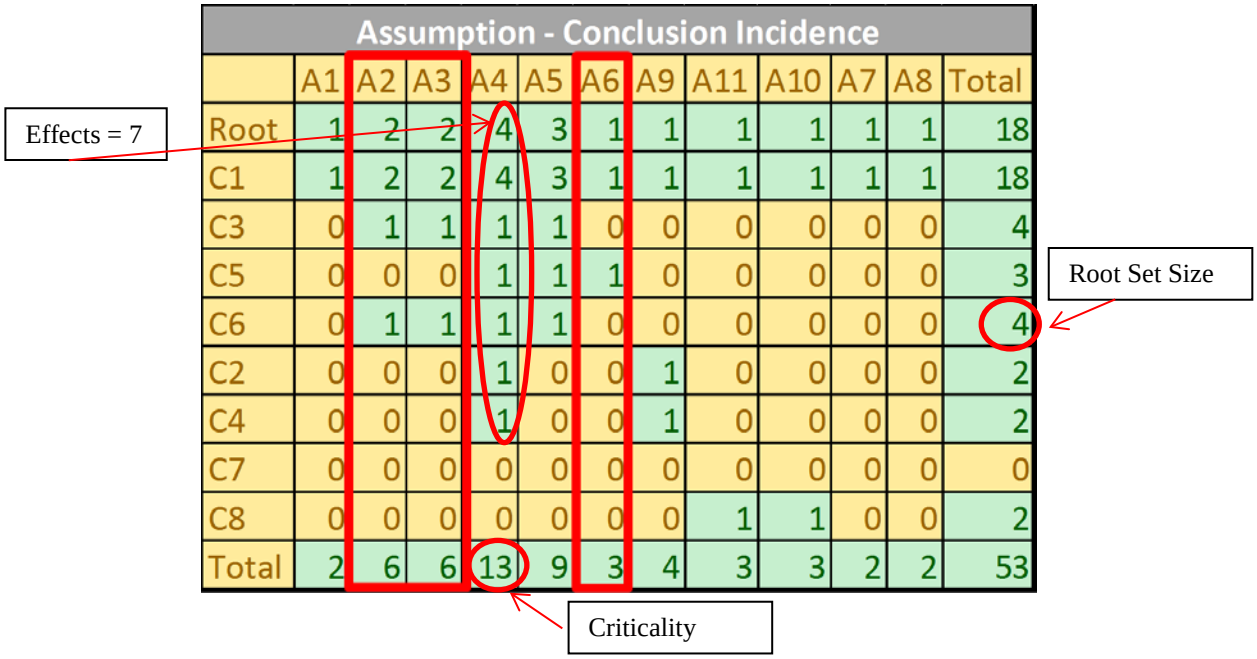
Initially, the proof efforts started with the low-level semantics, with the goal of proving properties to support the assurance case of the hotel, such as "the previous guest no longer can access a room once a new guest enters". The low-level semantic theorems, like the proofs in the computer science domain, did not require assumptions to make the proof successful.

One application level theorem proved that the previous key could not open the lock once a successor key had opened it. This proof required the assumption that the two keys were not duplicates, where the previous and current values for both keys were all set to the same value. This came up as a counter-example from ACL2. The proof also required the assumption that the keys were not inverse of each other, that is, that each key's previous value was equal to the other key's current value. Figure 1 shows the assurance case fragment for the proof and its dependencies on assumptions A2 and A3 (as processed and imported into the combined SITAPS/Certware suite).

Figure 1 The assurance case fragment for the previous key proof requires two assumptions to be met.

After integrating that proof into the assurance case, further analysis showed other claims depended upon the same two assumptions, that successive keys are not equal and not inverses. Table 1 shows the SITAPS derived metrics of the assurance case, where assumptions A2 and A6 represent *the keys are not duplicate* and A3 that *the keys are not inverse*. A2, A3, and A6 have a combined criticality of 15.

Table 1 Initial assurance case metrics for the hotel example.



	A1	A2	A3	A4	A5	A6	A9	A11	A10	A7	A8	Total
Root	1	2	2	4	3	1	1	1	1	1	1	18
C1	1	2	2	4	3	1	1	1	1	1	1	18
C3	0	1	1	1	1	0	0	0	0	0	0	4
C5	0	0	0	1	1	1	0	0	0	0	0	3
C6	0	1	1	1	1	0	0	0	0	0	0	4
C2	0	0	0	1	0	0	1	0	0	0	0	2
C4	0	0	0	1	0	0	1	0	0	0	0	2
C7	0	0	0	0	0	0	0	0	0	0	0	0
C8	0	0	0	0	0	0	0	1	1	0	0	2
Total	2	6	6	13	9	3	4	3	3	2	2	53

Through the scoping of our problem space, we have the closed system assumption that all keys come from the same key encoder at the hotel's front desk. Thus, by proving that the key encoder does not produce duplicate or inverse successive keys, we can eliminate assumptions A2, A3, and A6. We add claims C9 and C10, with their proofs as evidence, to discharge these assumptions, as shown in Figure 2.

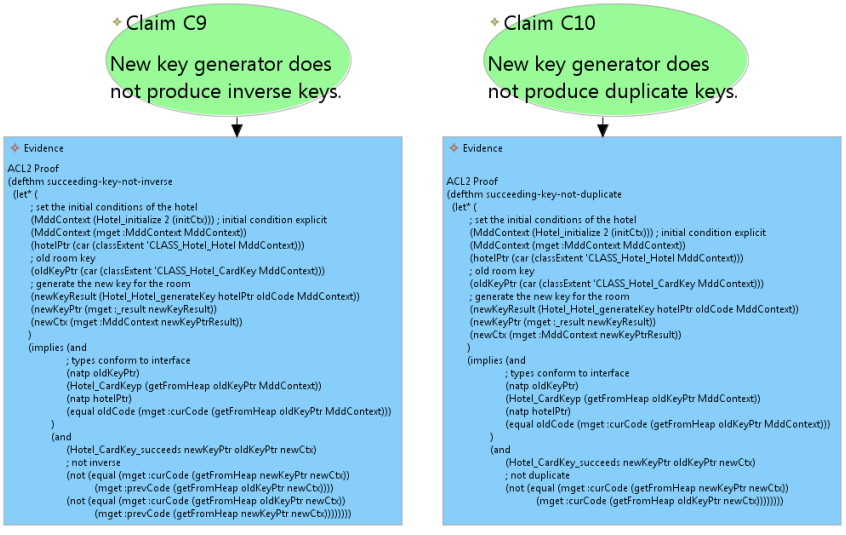


Figure 2 New proof discharge assumptions about key encoding.

These proofs allow us to remove the assumptions from the assurance case, resulting in the SITAPS derived metrics shown in Table 2. A2, A3, and A6 have been removed, and replaced with conclusions C9 and C10 from the new proofs, without additional assumptions (beyond the

scoping). This effort removed 15 assumption-conclusion dependency paths from the overall assurance case graph.

Table 2 SITAPS metrics for the hotel example after introduction of new proofs to eliminate 3 assumptions.

Assumption - Conclusion Incidence									
	A1	A4	A5	A9	A11	A10	A7	A8	Total
Root	1	4	3	1	1	1	1	1	13
C1	1	4	3	1	1	1	1	1	13
C3	0	1	1	0	0	0	0	0	2
C5	0	1	1	0	0	0	0	0	2
C6	0	1	1	0	0	0	0	0	2
C2	0	1	0	1	0	0	0	0	2
C4	0	1	0	1	0	0	0	0	2
C7	0	0	0	0	0	0	0	0	0
C8	0	0	0	0	1	1	0	0	2
C10	0	0	0	0	0	0	0	0	0
C9	0	0	0	0	0	0	0	0	0
Total	2	13	9	4	3	3	2	2	38

For complete documentation, it would be useful to show both the initial assumptions and the proof that discharges those assumptions. Unfortunately, GSN notation within Certware does not support a construct for showing that an argument or evidence discharges an assumption.

If the lock and key encoder components were developed by different companies, using different verification tools, then integrating them through the assurance case may be the easiest approach. If both are in the same language, such as ACL2, we can combine these two proofs into one larger proof[11], showing that if key2 is generated by the key encoder as the successor to key1, then key1 will not open the door once key2 is used. Thus the assurance case becomes a living document that evolves with our understanding of the system, and can guide further verification efforts.

Assurance Case

The assurance case for the subset of the hotel that we have modeled is rooted in the claim that the guest and his possessions are secure in his room. A subclaim is that the current occupant is the only one that can open the door, and this breaks down into claims and arguments about the protocol for issuing new keys and transitioning the locks to new occupants. Preventing the

[11] Though in either case, care must be taken to ensure that the context of the assumption matches the context of the proof.

previous occupant from accessing the room becomes a primary subclaim we address with a proof of the key management protocol across three time periods - before check-in, during occupancy, and after checkout.

At this point, we scope our pedagogical example model with the following assumptions:

- We focus on the keys, locks, and the key encoder at the front desk
- We ignore physical attacks on the door and lock
- We assume that all keys are created by the front desk encoder
- We ignore the master key and assume it is physically secured by hotel personnel.
- We ignore cyber-attacks that may either invalidate assumptions or change the implementation from that which is modeled. The first issue can be addressed by SITAPS via specification improvement. The second issue is what motivates the work addressed at assurance in depth (no verification approach can survive arbitrary changes to the analyzed executable system)[12]

Figure 3 shows the assurance case with the root claim, subclaims (green ovals), supporting arguments (blue boxes), evidence - where proofs are considered evidence (blue boxes), and assumptions (light blue ovals).

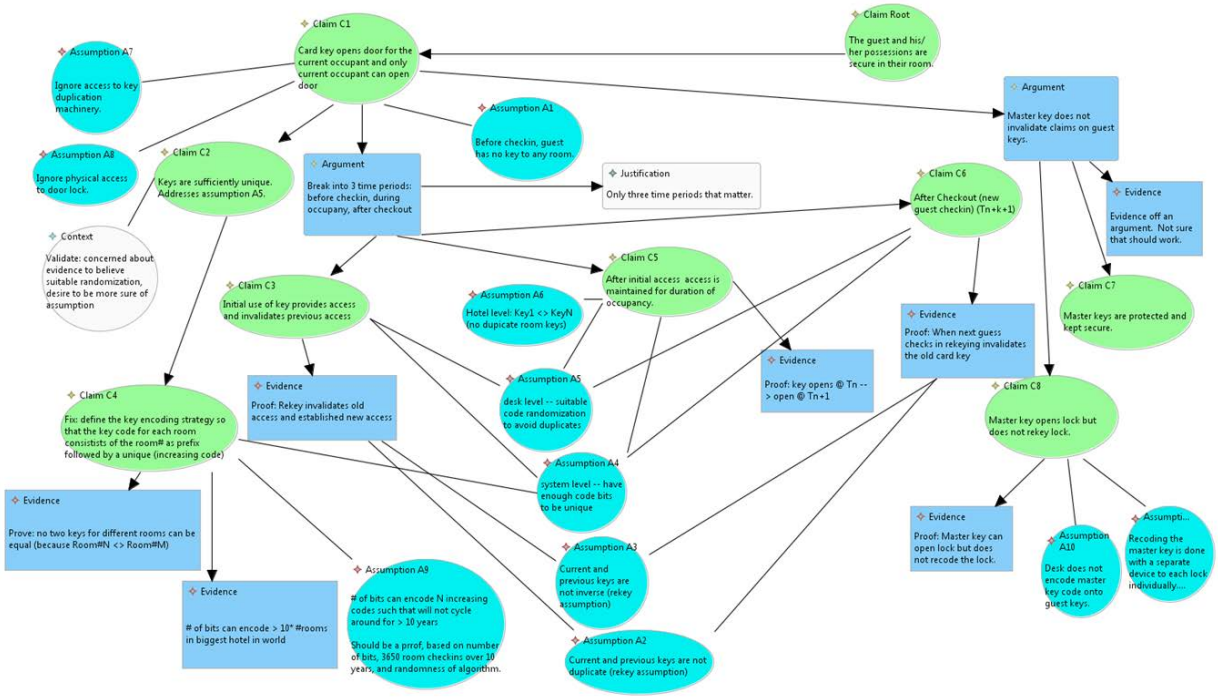


Figure 3 The assurance case supports the claim that the guest is secure in his room.

[12] While we ignore cyber-attacks for the purposes of modeling and evaluating the SITAPS approach, we have modeled part of the attack surface of the front desk's computer/key encoder for later analysis.

Integrating Proofs with Assurance Cases

Goal Structuring Notation is a general representation for capturing assurance cases. It models argument claims as goals and subgoals which are supported by evidence in the form of solutions. The type of argument is documented by a strategy and the strategy (or the goal itself) can be qualified by an explanatory context. The notation also supports ungrounded assumptions. Strategies can be further supported by justifications. In summary the elements of GSN are: goals, solutions, strategies, contexts, assumptions, and justifications.

The GSN meta-model is general enough to support many different argumentation styles and formalisms. Some work on assurance cases advocates specific argumentation styles, e.g., Toulmin structures as used in moral and legal reasoning and a host of other rhetorical styles (see, e.g., "Thank You for Arguing," Jay Heinrichs, 2007).

For our purposes, GSN as supported by the Certware Tool is adequate for capturing an integrated system assurance case consisting of:

- semi-formal argumentation that provides confidence in overall assurance goals
- stub representations of formally proved claims (intermediate proof structure elided)
- complete representation of formally proved claims (with intermediate chain of reasoning captured)

SITAPS integrates proofs into assurance cases using a claim node to represent a conclusion, an evidence node to represent the proof, and assumption nodes to represent the proof's substantive assumptions. The integrated assurance case forms a semi-formal proof structure which can be represented and analyzed in the SITAPS proof structure database.

Metrics Tools

SITAPS uses graph analysis to evaluate the proof structure and assurance case inputs, providing feedback on the use of assumptions. SITAPS counts the paths between each assumption and assurance case claim or proof conclusion to produce metrics on the inputs.

The SITAPS metrics tool is built on a Neo4j graph database. It uses the Cypher language to represent the graph inputs, creating the nodes and edges. SITAPS then uses the Neo4j Java API to traverse the graph from each assumption to each conclusion. In addition, it optionally produces a Graphviz file for visualization. Graphviz is a scalable open-source visualization tool for graphs.

SITAPS includes, as a separate component, Java programs to transform Certware assurance case XML files, in the form of .CAZ files, to Cypher programs that the SITAPS metrics tool operates on. The Resolute tool, part of Rockwell-Collins AGREE toolchain, also exports the .CAZ file format for the assurance cases it derives from AADL models. Note that an updated version of Certware is in progress and file formats may change from this version. SITAPS supports CERTWARE version 1.2.3.

The SITAPS User's Manual contains the detailed descriptions of the tools and interfaces

4.2 Verifying Domain Specific Language Specifications[13]

Ivory is a DSL in Haskell for embedded programming developed under the DARPA HACMS program. Ivory has semantics similar to C, but also provides memory safety, which is enforced by the Ivory type system. To capture design intent, Ivory has user specified assertions and procedure contracts. In addition, the Ivory compiler generates assertions to guard a program against a host of runtime violations including floating point exceptions, numerical overflows, index casting, and unbounded loops.

Verification of these assertions is crucial for two reasons. First, assertions are still runtime checks and failures of such are equivalent to uncaught exceptions (think Ariane 5). This importance cannot be underestimated in HACMS, since an Ivory autopilot will be flying a real helicopter with a safety pilot on-board. Second, runtime checks have runtime overhead: if these checks and their associated logic can be safely removed, memory consumption and execution time are reduced; important for embedded systems, which often run under tight resource constraints.

To address these concerns, we created the Dove system (a DSL Operational Verification Environment) to aid the verification of programs in imperative DSLs, and Ivory in particular. Like Ivory, Dove is embedded in Haskell, making it convenient for language translation since both Abstract Syntax Trees (ASTs) are represented as Haskell datatypes. The Dove system provides all the constructs for the Dove language as well as the Verification Condition (VC) generator, the Dove optimizer, and the interface to the backend prover, ACL2[14].

Interprocedural Verification and Runtime Check Optimization with Dove

In Dove, Ivory program verification is taken one procedure at a time. Starting at a procedure's arguments, the Dove verifier traverses the procedure's body generating verification conditions for assertions and procedure contracts along the way. To optimize-out proven checks, the verifier maintains a working copy of the procedure's AST. When an assertion or post-condition check is verified, the runtime check is removed. After verification, the modified AST is passed to a conventional optimizer and code generator.

During the traversal, the verifier accumulates a database of lemmas to aid the verification of future checks in a procedure. These include pre-conditions (requires) on procedure arguments and any prior check performed on a given branch, regardless of whether the check was verified. Checks that fail to prove remain in the generated code as do procedure pre-conditions; the later to avoid potential issues with recursive procedures. To help scale to global program verification, procedure calls are abstracted with the callee's procedure contracts. Specifically, the callee's pre-conditions (requires) are asserted and the post-conditions (ensures) are added to the lemma database.

---

[13] Reported in more detail in CDRL A008: Dove and Ivory: Verifying One DSL with Another

[14] Eakman, Greg, et. al., "Practical Formal Verification of Domain-Specific Language Applications", Proceedings, 7th Annual NASA Formal Methods Symposium, Pasadena, CA, April 27-29, 2015

Dove is a declarative, side effect free language, whose top level constructs are expressions. Ivory is not translated to an equivalent Dove program, but rather individual Ivory checks are translated into Dove expressions to form the associated VC for proof in ACL2.

The translation from Ivory to Dove VCs and then to ACL2 will be illustrated by way of an example. Assume we have a control system with two software actuated valves: valve A and valve B. The system has a safety property that states the two valves cannot be open at the same time. The Ivory function (openValveA) shown in Figure 4 is written in Ivory's DSL syntax embedded in Haskell. It provides the means to command valve A open. As inputs, openValveA takes two references that represent the states of both valve A and B (true means open). To adhere to the safety property, a pre-condition on openValveA requires that valve B must first be closed. The post-conditions state that openValveA will result in valve A being open and valve B remaining closed. An additional pre-condition is needed to ensure that the two state references are different. If they were the same, the function would result in both valves being open, which would obviously fail the post-condition requirement.

```
-- Command valve A open.
openValveA :: Def ('[Ref s (Stored IBool), Ref s (Stored IBool)] :-> ())
openValveA = proc "openValveA" $ \ valveOpenA valveOpenB ->

  -- Require that valve B must first be closed.
  requires (checkStored valveOpenB $ iNot) $

  -- Require that valveOpenA and valveOpenB are different references.
  requires (refToPtr valveOpenA /=? refToPtr valveOpenB) $

  -- Ensures that valve A is opened.
  ensures (const $ checkStored valveOpenA id) $

  -- Ensures that valve B remains closed.
  ensures (const $ checkStored valveOpenB iNot) $

  body $ do
    -- Open valve A.
    store valveOpenA true
    retVoid
```

Figure 4 Ivory Function open ValveA.

During the verification traversal of Ivory procedures, Dove generates and checks VCs in order. Because there are no internal assertions in this example, the first VC for verification is the post-condition that ensures valve A will be opened on return from the procedure. Figure 5 shows this VC translated to Dove. Prior to translating to ACL2, Dove optimizes the VC to that shown in Figure 6. Dove optimizations consist of a combination of constant propagation, expression inlining, and null effect removal. At this point, Dove is then translated to ACL2 as shown in Figure 7, which ACL2 easily verifies. To ease both ACL2 program generation and execution, we created a Haskell DSL and interface for ACL2.

```
-- Create the initial stack.fora
ll free0 in
let stack0 = free0 in

-- Assume the initial stack is an array type.
let assume0 = (true implies (isArray stack0)) in

-- Create the initial environment from the procedure's
-- arguments, i.e. the two valve state references.
forall free1 in
forall free2 in
let env0 = {var0 = free1, var1 = free2} in

-- Assume the two arguments are references, i.e. they are
-- integer types and they are within the bounds of the stack.
let assume1 = (true implies (isInt env0.var0)) in
let assume2 = (true implies (env0.var0 ge 0)) in
let assume3 = (true implies (env0.var0 lt (arrayLength stack0))) in
let assume4 = (true implies (isInt env0.var1)) in
let assume5 = (true implies (env0.var1 ge 0)) in
let assume6 = (true implies (env0.var1 lt (arrayLength stack0))) in

-- Assume the first pre-condition, i.e. valve B is closed.
let env1 = (overlay {pre0 = stack0[env0.var1]} env0) in
let assume7 = (true implies (not env1.pre0)) in

-- Assume the second pre-condition, i.e. valve A and B are
-- different references.
let assume8 = (true implies (not (env1.var0 eq env1.var1))) in

-- Update the valve A state, i.e. set it to open (true).
let stack1 = (update env1.var0 true stack0) in

-- Define the first VC, i.e. that valve A is open.
let env2 = (overlay {pre1 = stack1[env1.var0]} env1) in
let vc0 = (true implies env2.pre1) in

-- Construct the final check with all the assumptions included.
(((((((((true and assume0) and assume1) and assume2) and assume3)
    and assume4) and assume5) and assume6) and assume7) and assume8)
    implies vc0)
```

Figure 5 Dove VC for First Post Condition of openValveA.

```
forall free0 in
forall free1 in
forall free2 in
((((((((((isArray free0) and (isInt free1)) and (free1 ge 0))
  and (free1 lt (arrayLength free0))) and (isInt free2))and (fr
  ee2 ge 0)) and (free2 lt (arrayLength free0)))and (not free0[
  free2])) and (not (free1 eq free2)))
  implies (update free1 true free0)[free1])
```

Figure 6 Optimized Dove VC.

```
(thm
(implies
    (and
        (consp free0)
        (integerp free1)
        (>=free1 0)
        (< free1 (len free0))
        (integerp free2)
        (>=free2 0)
        (< free2 (len free0))
        (not(nth free2 free0))
        (not(equal free1 free2))
        )
    (nth free1 ( update-nth free1 t free0))
    ))
```

Figure 7 Optimized Dove VC Translated to ACL2.

To assist in debugging the Dove system during development and to gauge Dove's performance of verification, a set of test cases were constructed. Many of these tests were designed to target specific areas of the Ivory language, while others tried to represent real world programming scenarios. In one test suite that produced 53 VCs, 47 were verified in a total time of 1.3 seconds. In a more challenging test suite, where not all assertions where designed to pass, 4 VCs out of 8 were verified in 37 seconds. In our experience with these test cases, there seems to be two common outcomes. Either a VC returns quickly from ACL2 (either pass or fail) or the VC causes ACL2 to loop forever. Of the cases where a VC causes ACL2 to loop endlessly, we have identified that this is sometimes due to how the initial stack is modeled. By using Dove's default model of the initial stack as a free variable (i.e., on entrance to the procedure the stack could have any number of elements with any value) then ACL2 will not converge in these cases. However, if the initial stack is either empty or has finite length, ACL2 is able to return. In one particular case for a VC known to be true, specifying a non-zero, finite length stack caused ACL2 to converge but failed to return a proof. If for the same test case the initial stack was finite length with concrete values, ACL2 was able to both quickly return and verify the VC. This situation has cropped up several times during Dove development and warrants further investigation.

5 Conclusions and Recommendations

The use of proof in HACMS components and systems was different than expected as compared to the use of proof in mathematics. For example, consider the proof of the Pythagorean Theorem illustrated at: http://www.cut-the-knot.org/pythagoras/ where 112 different proofs are provided. These proofs meet the social obligation of proof as providing a vehicle for both proof and persuasion of a candidate theorem. The proofs are reviewable and confirmable. In general, the proof tools used for formal reasoning about software artifacts do not have this property. The proof in such cases is typically provided by a successful application of the reasoning tool, often guided by a proof outline (tactics). The evidence of success is largely that the proof assistant terminates with a positive result.

As long as proof assistants rely on this correct-by-construction approach, it will be difficult to either compose independently verified components or validate system configurations of multiple components. Software verification tools need to adopt approaches akin to proof-carrying code to capture the assumptions and domain models used in producing proofs of correctness. This will allow analysis of the composition of components and detection of inconsistent underlying assumptions and models.

The challenge of addressing the software semantics problem is to introduce proof into existing, large-scale developments in such a way that cost is commensurate with reward: keeping in mind that this ratio is based on the necessity of runtime correctness. Possible avenues that might be explored in a High Assurance Software Development program include:

- Integrate proof techniques into a dialect of an existing mainstream language, for example: "high-assurance Java", based on refinement types, and theorems that generate as tests until fully proven;
- Build a bridge between a full proof environment, like Coq, and, e.g., the JVM, allowing the 1% of mission critical code to be fully verified, while being easily integrated with the remaining 99%;
- Integrate proof tools with software development environments to provide the feedback necessary to restructure complex software to accommodate proof.

The desired end-state includes the ability to ensure that important behaviors exist in the final product, and that important misbehaviors do not exist. Further, to have an impact in current engineering organizations, the ability to deliver a running, testable system must be preserved at all stages.

What makes formal methods essential is the ability to answer these questions in a provable way, rather than either the statistical assurance of testing—where the choice of inputs is subject to human bias—or reliance solely on oversight and review. However, the artifacts that contribute to the development of formal proofs, i.e., the underlying non-executable models and assumptions, must themselves become part of the analyzable system "code" base in order to support system validation and composition processes.

LIST OF ACRONYMS, ABBREVIATIONS, AND SYMBOLS

ACL2	A Computational Logic for Applicative Common Lisp
AST	Abstract Syntax Tree
DOVE	A DSL Operational Verification Environment
DSL	Domain Specific Language
GSN	Goal Structuring Notation
HACMS	High-Assurance Cyber Military Systems
SAFE	Semantically Aware Foundation Environment
SITAPS	Specification Improvement through Analysis of Proof Structure
VC	Verification Condition